Sissejuhatus infotehnoloogiasse

Programmeerimiskeeled

Ülevaade loengust

- Meeldetuletus eelmistest loengutest: protsessor ja assembler ja andmestruktuurid
- Assembler ja C keel ja kompileerimine ja linkimine ja käivitamine
- Kompileerimine vs interpreteerimine
- Esimesed kolm programmeerimiskeelt
- Programmeerimiskeelte suur pilt
- Populaarsused
- Keelte põhiideed ja näited
- Kiirused

Meie näiteprogramm: summeerime arve 1 .... N

```
int main(int argc, char **argv) {

  int n=10;
  int i,sum=0;

  printf("Tere!\n");

  for(i=0; i<=n; i=i+1)
    sum = sum + i;

  printf("summa on %d\n",sum);
}
```

Sumto ja Intel 386, 486, Pentium, ...

- 386 on vähe registreid, argument saadetakse hariliku mälu kaudu. Resultaat saadetakse registris %edx.

```
_sumto:
        pushl %ebp              ; Loome ''framepointer''-i
        movl  %esp,%ebp         ;
        movl  8(%ebp),%ecx      ; Võta n.
        xorl  %eax,%eax         ; sum = 0
        xorl  %edx,%edx         ; i = 0
        cmpl  %ecx,%eax         ; Kui i>n ...
        jg L3                   ; ... mine L3
        .align 2
L5:     addl  %edx,%eax         ; sum = sum + i
        incl  %edx              ; i = i+1
        cmpl  %ecx,%edx         ; Kui i<=n ...
        jle L5                  ; ... mine L5
L3:     leave                   ; Taasta ebp.
        ret                     ; Valmis!
```

Sumto ja ARM

- ARMil on 16 registrit r0, ..., r15, kuid osa neist on eritähendusega (näiteks, r15 on program counter)

```
        entry
sumto:
        mov r1, #0              ; sum = 0
        mov r2, #0              ; i = 0
        ldr r3, N               ; r3=N
loop:
        cmp r3, r2              ; eeldame r3=N
        bgt finish              ; kui n>i, mine finish
        add r2, r2, #1          ; i = i + 1
        add r1, r1, r2          ; sum = sum + i
        B loop                  ; uus tsükkel
 finish:
        end
N       DCD  #1000              ; mälupesa N algväärtusega
```

Uurime C kompilaatorit

- Võtame eelmise 0 ... N summa näite C keeles ja vaatame, mida kompilaator teeb.

Programmeerimiskeeled on inimeste jaoks!

- **Arvuti** ei saa programmeerimiskeelest midagi aru. Talle on vaja masinkoodi.

- Programmeerimiskeeled on selleks, et **inimestel** oleks lihtsam programme kirjutada ja nendest pärast aru saada.

Kõrgkeeled

- **Automatiseerivad ja lihtsustavad** hulga “harilikke” protseduure, mida assembleris programmeerides vaja
- Ei anna assembleriga analoogilist kontrolli masina üle
- Kõrgkeeled on erineva abstraktsusastmega:
  - Masinalähedane ja ebamugav: Fortran, C (portaabel assembler)
  - Abstraktsem ja mugavam: Lisp, Ada, ML, Java, Python,
- **Peale programmeerimiskeelte on veel hulk muid keeli:**
  - Päringukeeled (SQL, RDQL, ...)
  - Kujunduskeeled (HTML, PS, ...)
  - Spetsifitseerimiskeeled (loogikakeeled, UML, ...)
  - ....

Kompileeritava programmi valmimine

Olgu meil (näiteks C keeles) failid main.c ja swap.c
Teeme gcc main.c swap.c -o minuprogramm

Kompilaator (näiteks gcc) teeb järjest mitut eri asja:

- **Kompileerimine**
  - Kompilaator teeb neist assemblerikeelsed ajutised failid
  - Kompilaator teeb assemblerfailidest masinkood+sümbolinfo failid

- **Linkimine**
  - Linkur otsib kokku vajalikud olemasolevad failid osa sümbolinfo seostamiseks päris koodi-viidetega

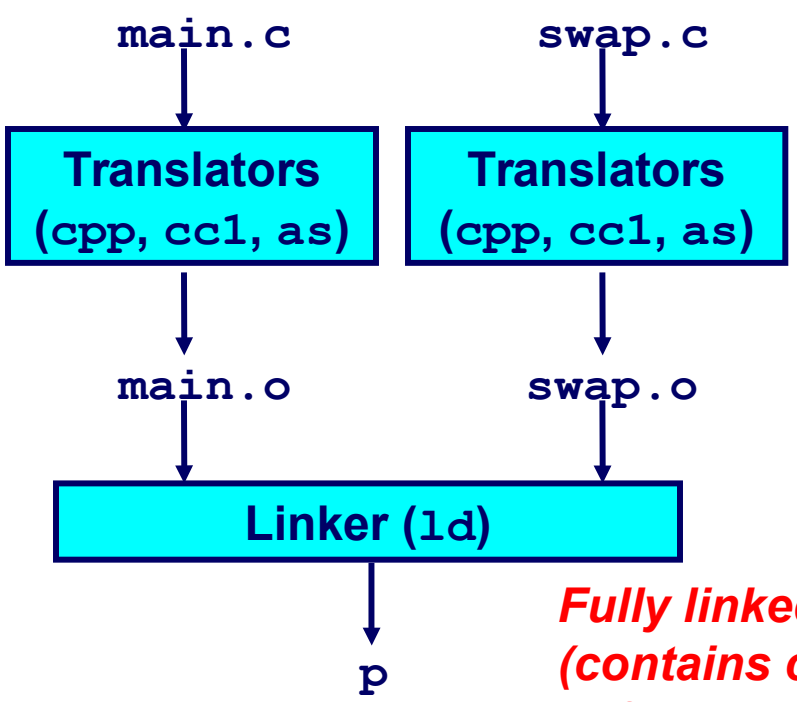
Käivitame saadud programmifaili minuprogramm:

- Opsüsteemi **loader** otsib lisaks vajalikud olemasolevad failid osa sümbolinfo seostamiseks päris koodi-viidetega
- Saadud kogum paigutatakse mällu, tehakse opsüsteemi infoblokk tema jaoks (protsess) ja kogum käivitatakse

(CMU Bryant & Hallaron course) Static Linking

Programs are translated and linked using a *compiler driver*:

- `unix> gcc -O2 -g -o p main.c swap.c`
- `unix> ./p`

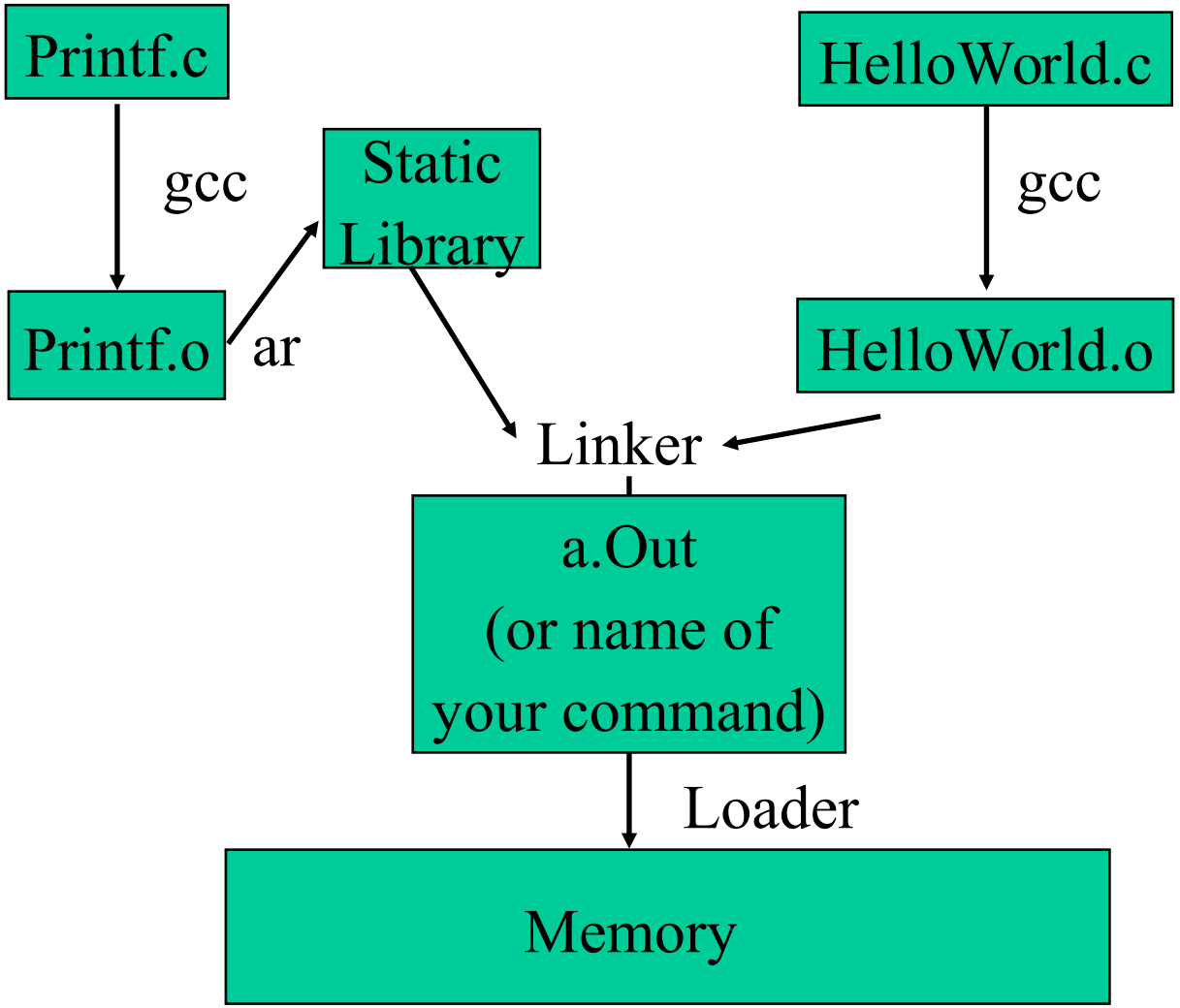


Source files

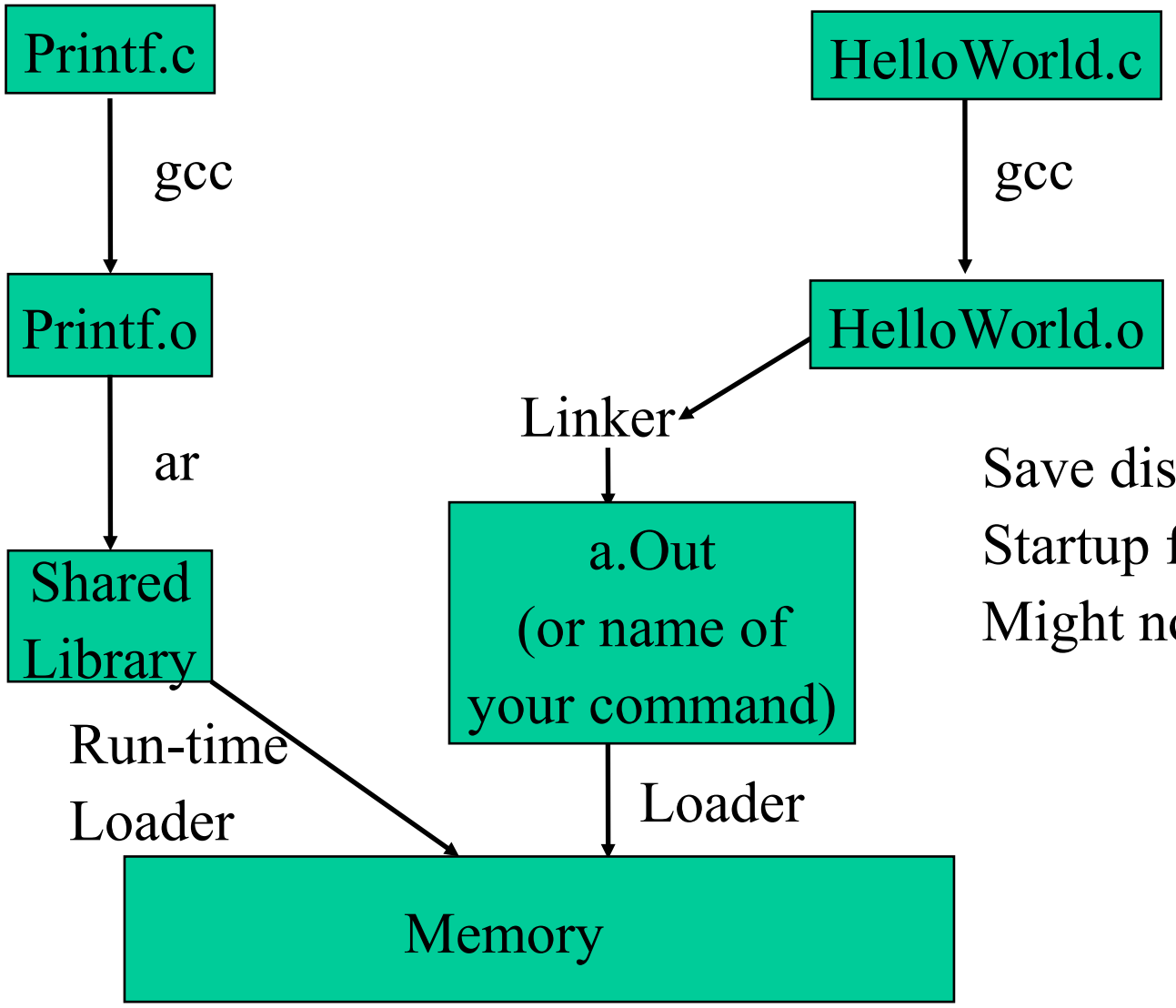
Separately compiled relocatable object files

Fully linked executable object file (contains code and data for all functions defined in main.c and swap.c

(CMU B&H ..) Static Linking and Loading



(CMU B&H ..) Run-time Linking/Loading

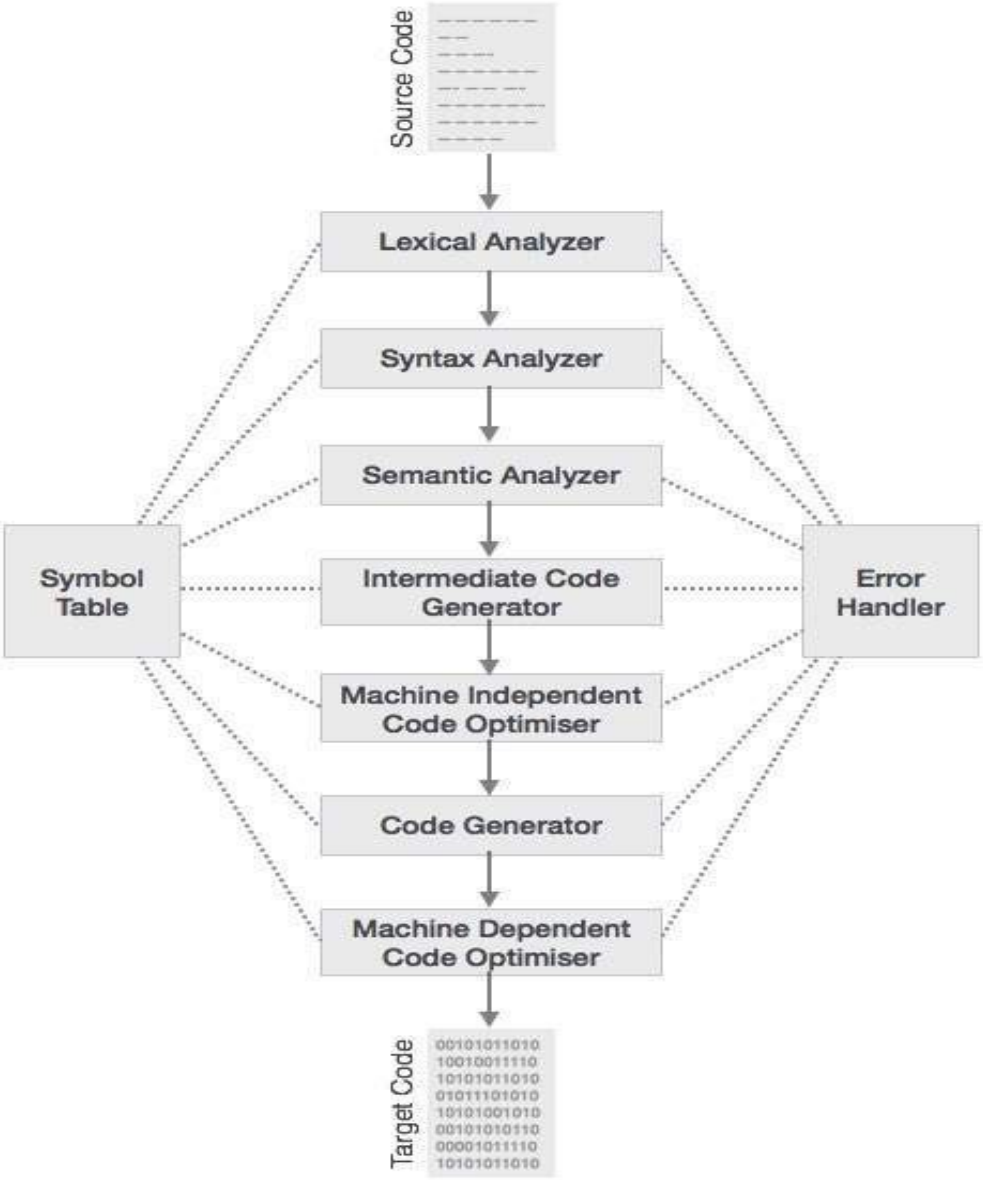


Save disk space.

Startup faster.

Might not need all.

Kompileerimise faasid



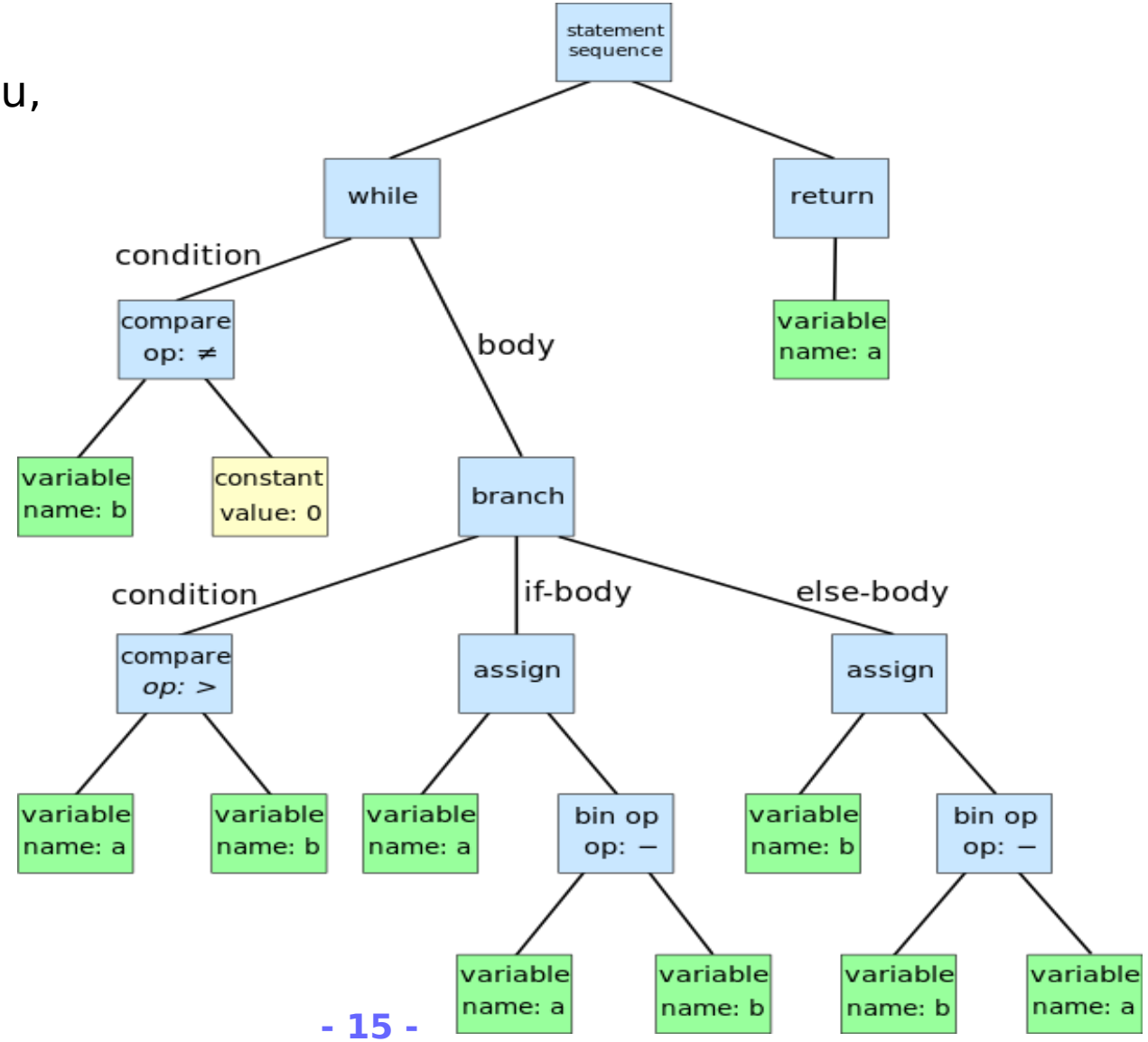
Parser ehk süntaksianalüüs: näitesisend

Olgu meil programmijupp

```
while b ≠ 0
      if a > b
         a := a − b
      else
          b := b − a
 return a
```

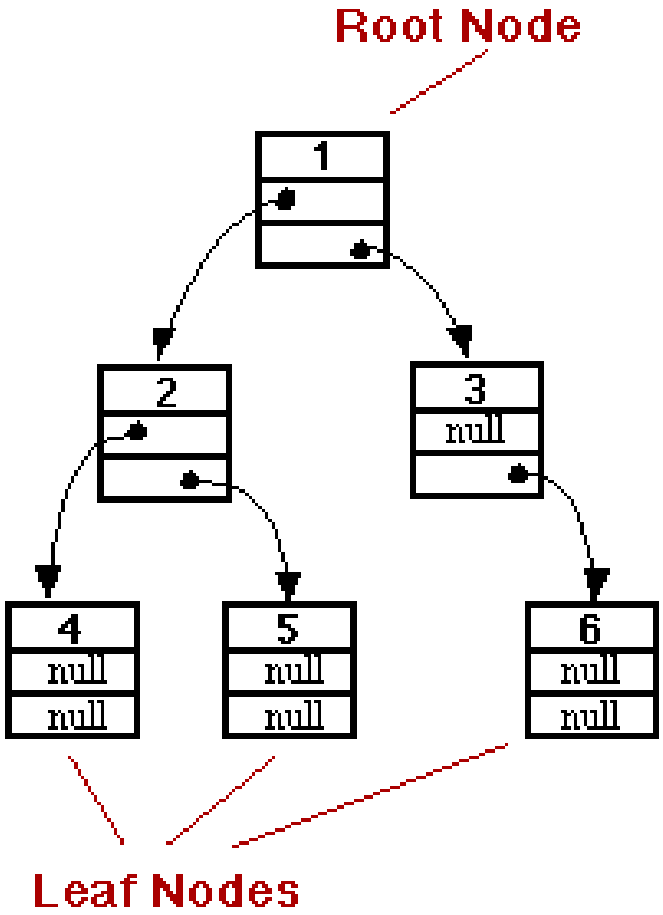
Parser ehk süntaksianalüüs: süntaksipuu

Parser ehitab mällu süntaksipuu, mida siis edasi töödeldakse



Meeldetuletus: puud mälus

Siin näitepuus koosneb iga element väärtusest ja kahest mäluaadressist. Null tähistab aadressi 0, mida normaalselt aadressina ei kasutata.



Kuidas keeles X kirjutatud programmi täidetakse?

Kaks põhivarianti keeles X programmi täitmiseks.

- **Kompileerimine**: masinkoodis programm nimega kompilaator teisendab keeles X programmi masinkoodfailiks Y. Seejärel täidetakse saadud masinkoodis programm Y. Näited: C, Fortran, Go.

- **Interpreteerimine**: masinkoodis programm nimega interpretaator loeb sisse X keeles faili, teisendab ta nö pseudo-assembleriks / vahekoodiks ja asub seda vahekoodi varianti rida-realt täitma. Näited: Python, PHP, Perl, vanemad Javascripti mootorid jne.

NB!

- Programmi interpreteerimine on ca 10-200 korda aeglasem, kui kompileeritud koodi täitmine.
- Põhimõtteliselt saaks igas keeles kirjutatud programme nii interpreteeritult täita kui kompileerida.
- Praktikas eelistatakse vahel interpreteerimist, vahel kompileerimist.

Vahekood Pythoni näitel

Olgu meil väike näiteprogramm

```
a, b = 1, 0
if a or b:
    print "Hello", a
```

Vahekood Pythoni näitel

Vahekood kommentaaridega

```
code   6404005c02005a00005a0100650000700700016501006f0d000164020047650000474 86e01000164030053
  1           0 LOAD_CONST               4 ((1, 0))
              3 UNPACK_SEQUENCE          2
              6 STORE_NAME               0 (a)
              9 STORE_NAME               1 (b)

  2          12 LOAD_NAME                0 (a)
             15 JUMP_IF_TRUE             7 (to 25)
             18 POP_TOP
             19 LOAD_NAME                1 (b)
             22 JUMP_IF_FALSE           13 (to 38)
        >>   25 POP_TOP

  3          26 LOAD_CONST               2 ('Hello')
             29 PRINT_ITEM
             30 LOAD_NAME                0 (a)
             33 PRINT_ITEM
             34 PRINT_NEWLINE
             35 JUMP_FORWARD             1 (to 39)
        >>   38 POP_TOP
        >>   39 LOAD_CONST               3 (None)
             42 RETURN_VALUE
```

Kuidas keeles X kirjutatud programmi täidetakse?

Interpreteerimise kompromissvariandid:

- Kompilaator kompileerib X faili **vahekoodiks** Y, seejärel interpreteeritakse vahekoodi Y (Python, Java).

- Interpretaator interpreteerib vahekoodi Y, kuid **kompileerib töö ajal osa Y-st masinkoodiks**, mida seejärel täidab (Java, C#, Firefoxi Javascript) nn just-in-time compilation ehk **JIT**.

- Chrome V8 Javascript kompileerib algul kogu programmi masinkoodiks kiire kompilaatoriga, seejärel kompileerib töö käigus selgunud kriitilised kohad aeglasema optimeeriva kompilaatoriga, mis annab kiiremini töötava tulemuse.

Programming and description languages

- **Programming languages**, in the familiar sense:

  Fortran, C, Java, C#, Python, Javascript etc.

- **Description languages** (i.e. not for programming):

  Text layout: html
  Html layout nuances: css
  Database query tasks: SQL
  Data representation: XML, json, csv
  ... etc

Miks nii palju keeli?

- Eri valdkondade / eesmärkide jaoks ei ole ühte ideaalset keelt
- Programmeerijate eelistused on erinevad
- Vaata ka DSL (domain specific language):

  https://en.wikipedia.org/wiki/Domain-specific_language

Keelte erisused: kolm põhiasja

- **Süntaks** (kuidas kirjutatakse näiteks **if .. then .. else** ühes või teises keeles)
- **Semantika** ehk tähendus (mida õigesti kirjutatud programm tegelikult siis teeb)
- **Teegid (libraries)** (millised **valmisprogrammijupid** on selle keele jaoks kergesti kättesaadavad või kohe kaasa pandud)

Keeled: tüüpilised asjad, mida pea iga keel pakub

- **Primitiivsed andmetüübid:**
  - int, char etc (näiteks: 1 ja –3 on int-id, ‘c’ ja ‘a’ on char-id)
  - string (näiteks “aaa123bb”)
  - massiiv (näiteks a[1]=2; a[2]=20; a[3]=15; y=2; x=a[y] +a[1]+3;)
- **Avaldised:**
  - näiteks x = (y*2) – (5+x);
- **Elementaarsed juhtkonstruktsioonid:**
  - valik: if ... then ... else
  - tsükkel: while(x<10) x=x+1;
- **Funktsioonid:**
  - defineerime: int kuup(int x) { return x * x * x}
  - kasutame: x = kuup(1+kuup(3))+kuup(y);
  - kasutame rekursiivselt:

int fact(int x) { if (x<=0) return 1; else return x*(fact(x-1)); }

Keeled: näited lisavõimalustest eri keeltes

- Kiired bitioperatsioonid, otsepöördumine mälu kallale: C
- Keerulisemad andmetüübid: listid, hash tabelid jne: Lisp, Python, Javascript
- Erikonstruktsioonid stringitöötluseks: Perl, PHP

- Objektid: C++, Java, C#, Python, Lisp
- Moodulid (enamasti ühendatud objektidega): C++, Java, C#
- Veatöötluse konstruktsioonid (exceptions): Python, Java, C#

- Prahikoristus: kasutamata andmed visatakse välja (Java, Python, Lisp, ...)
- Sisse-ehitatud tugi paralleelprogrammide jaoks: Java, C#
- Reaalaja-erivahendid: Ada

- “Templates” (programm tulemuse sees): PHP, JSP, Pyml
- Uute programmide konstrueerimine töö käigus: Lisp, Scheme, Javascript
- Loogikareeglid: Prolog
- “laisk” viis funktsioone arvutada: Miranda, Hope, Haskell
- Pattern matching (viis funktsioone defineerida): ML, Haskell
- **jne...**

Fortran: summeeri arve 0...n

- FORTRAN

```
      INTEGER FUNCTI0N sumto(n)
      isum = 0
      DO i 10 = 0,n
      isum = isum + i
10    CONTINUE
      sumto = isum
      RETURN
      END
```

COBOL: summeeri arve 0...n

- COBOL

```
PROCEDURE SUMTO USING N, Answer.
Begin.
  PERFORM VARYING LoopCount FROM 0 BY 1
    UNTIL LoopCount GREATER THAN N
      MULTIPLY Answer BY LoopCount GIVING Answer.
  END-PERFORM.
  EXIT PROGRAM.
```

LISP: summeeri arve 0...n

- LISP

```
(defun sumto (n)
   (if (= 0 n)
       0
       (+ n (sumto ( n 1))) ))
```

Sumto ja C

- C (ja C++ ja Java ja C#)

```
int sumto(int n) {
  int i,sum = 0;
  for(i=0; i<=n; i=i+1)
    sum = sum + i;
  return sum;
}
```

Sumto ja Modula-2

- Modula-2

```
PROCEDURE sumto(n:INTEGER):INTEGER;
VAR sum,i:INTEGER;
BEGIN
      sum:=0;
      FOR i:=0 TO n DO
        sum:=sum+i
      END;
      RETURN sum
END sumto;
```

Sumto ja Modula-2

- Modula-2

```
PROCEDURE sumto(n:INTEGER):INTEGER;
VAR sum,i:INTEGER;
BEGIN
      sum:=0;
      FOR i:=0 TO n DO
        sum:=sum+i
      END;
      RETURN sum
END sumto;
```

Sumto ja Ada

- Ada

```
function sumto(n: in INTEGER) return INTEGER is
    sum : INTEGER := 0;
begin
    for i in 0..n loop
       sum := sum + i;
    end loop;
return sum;
```

Sumto ja Python

- Python

```
def sumto(n):
  sum=0
  for i in range(n+1):
    sum=sum+i
  return sum
```

Pythoni vahekood: keerukamat lisalugemist

https://nedbatchelder.com/blog/200804/the_structure_of_pyc_files.html

http://infohost.nmt.edu/tcc/soft/pythondoc/lib/bytecodes.html

https://www.python.org/dev/peps/pep-0339/

https://arxiv.org/pdf/1306.6047v2.pdf